Engineer

Limitation de débit

plafond de requêtes par client pendant un intervalle donné

Ce que c’est

La limitation de débit plafonne le nombre de requêtes qu’un client peut envoyer à un service pendant un intervalle donné. Elle est appliquée par le service ou par un composant placé devant lui. Toute limite répond à trois questions : qui compte comme client, comment les requêtes sont comptées et ce qu’il advient de celle qui dépasse le seuil. Le code d’état HTTP prévu pour cette dernière question, 429 Too Many Requests, a été défini par la RFC 6585 en avril 2012, laquelle laisse délibérément les deux premières ouvertes. Un serveur peut compter par ressource, sur l’ensemble du serveur ou sur un groupe de serveurs, et reconnaître le client par ses informations d’authentification ou par un cookie avec état.

En pratique, la clé se choisit parmi un petit nombre de possibilités : l’adresse IP du client, le compte connecté, une clé d’API ou l’un de ces éléments combiné au chemin, afin qu’un formulaire de connexion et une page produit aient des limites différentes. Une requête au-delà de la limite reçoit l’une de trois réponses. Le serveur peut la refuser avec un code 429, de préférence accompagné d’un en-tête `Retry-After` indiquant le délai d’attente, soit en secondes, soit sous forme de date HTTP. Il peut retenir la requête et la traiter un peu plus tard, ce qui lisse une pointe plutôt que de la rejeter. Ou, lorsque le volume est lui-même l’attaque, il peut couper la connexion sans répondre. La RFC 6585 propose précisément cette solution, car rédiger un 429 poli pour chaque requête d’un déluge consomme aussi des ressources.

La méthode de comptage détermine le comportement d’une limite à ses frontières. Une fenêtre fixe compte les requêtes par tranches calées sur les minutes ou les heures de l’horloge et repart de zéro au passage à la tranche suivante. Elle est peu coûteuse et facile à expliquer, mais permet au client de dépenser toute sa réserve à la dernière seconde d’une fenêtre, puis de nouveau à la première de la suivante : avec une limite de 100 par minute, cela représente

200 requêtes en deux secondes. Une fenêtre glissante examine toujours la même durée en arrière à partir de l'instant présent : aucune frontière n'est exploitable. Les tables de suivi de HAProxy conservent de tels compteurs, et un exemple du manuel compte le rythme des connexions sur une fenêtre glissante de 30 secondes. Le seau percé utilisé par nginx laisse passer les requêtes à un débit moyen fixe et conserve une petite file pour celles qui arrivent plus vite, en refusant ce qui la ferait déborder. Un seau à jetons fonctionne à l'inverse : des jetons s'accumulent à rythme constant jusqu'à un plafond, chaque requête en dépense un, et un client resté silencieux peut dépenser sa réserve en une rafale.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que la limitation de débit ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des comptes, mots de passe et droits attribués selon les besoins exacts de chaque poste, puis retirés le jour du départ.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 1 000 règles RBAC pour sécuriser Linux — 1,000 règles d'accès écrites pour combler une faille de sécurité

1 000 règles RBAC pour sécuriser Linux Contactez-nous →

Pourquoi c'est important

Un point d'accès public ne choisit ni qui l'appelle ni à quelle fréquence. Un robot copiant un catalogue, un script pris dans une boucle de nouvelles tentatives, un attaquant essayant des mots de passe divulgués sur le formulaire de connexion et un partenaire dont le traitement nocturne a été écrit trop vite se ressemblent pour le serveur : davantage de requêtes qu'il n'a été dimensionné pour en traiter, provenant de quelques sources. Sans limite, le client le plus actif décide de la capacité laissée aux autres. Avec une limite, un client ne peut utiliser que sa part, et le reste du service conserve sa vitesse normale pendant qu'il attend.

Certaines requêtes coûtent beaucoup plus que d'autres. Une page produit peut être servie depuis un cache en une milliseconde, tandis qu'une recherche, un export ou un appel à une API externe payante consomme réellement du temps processeur ou de l'argent à chaque appel. Ce sont précisément les chemins visés par un utilisateur abusif, parce qu'ils ne lui coûtent rien. Une limite globale assez basse pour protéger l'export étoufferait la navigation ordinaire : une limitation utile est donc définie par chemin, chaque seuil étant choisi selon son coût et son usage légitime.

Une limite fait aussi partie du contrat d'une API. Les clients qui connaissent les règles peuvent adapter leur rythme sans jamais les atteindre, ce qui coûte moins aux deux côtés que de découvrir la limite par des refus. HTTP ne possède pas d'en-tête normalisé pour cela. De nombreuses API envoient `X-RateLimit-Limit`, `X-RateLimit-Remaining` et `X-RateLimit-Reset`, mais un même nom véhicule des valeurs différentes selon l'implémentation, comme le constate un projet de l'IETF visant à normaliser cette idée. Ce projet définit deux champs, `RateLimit-Policy` pour le quota et sa fenêtre et `RateLimit` pour ce qui reste ; sa onzième version est datée du 23 mai 2026 et, comme tout Internet-Draft, constitue un travail en cours plutôt qu'une norme.

À quoi cela ressemble en pratique

Dans nginx, une limite nécessite deux directives : l'une dans le bloc `http` pour créer le compteur, l'autre à l'endroit où il s'applique :

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=1r/s;
limit_req_status 429;

server {
    location /search/ {
        limit_req zone=perip burst=5 nodelay;
        proxy_pass http://app;
    }
}
```

La clé est l'adresse du client sous forme binaire, longue de 4 octets pour IPv4 et de 16 pour IPv6, et les compteurs résident dans une zone de mémoire partagée de 10 mégaoctets appelée `perip`. La documentation donne 128 octets d'état par clé sur une plateforme 64 bits : un mégaoctet contient donc environ 8 mille adresses et cette zone environ 80 mille. Lorsqu'elle est pleine, nginx évince l'entrée la moins récemment utilisée. `rate=1r/s` autorise une requête par seconde en moyenne. `burst=5` autorise jusqu'à cinq requêtes au-delà de ce débit à attendre leur tour plutôt que d'être refusées, et `nodelay` les sert immédiatement plutôt que de les espacer au rythme configuré. Tout dépassement de la rafale est rejeté. nginx répondrait par défaut à ce rejet avec 503, d'où la directive `limit_req_status` fixant 429. Les débits inférieurs à une requête par seconde s'écrivent par minute : `30r/m` correspond donc à une requête toutes les deux secondes. Avant l'activation d'une nouvelle limite,

`limit_req_dry_run on` continue à compter normalement et marque les refus qui auraient eu lieu par `REJECTED_DRY_RUN` dans la variable `$limit_req_status`, sans rien refuser.

Côté réception, un client qui atteint la limite devrait lire la réponse plutôt que réessayer aussitôt. Un 429 d'une API suivant le projet de l'IETF pourrait ressembler à ceci :

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/problem+json
Retry-After: 20
RateLimit-Policy: "default";q=100;w=60
RateLimit: "default";r=0;t=20
```

La politique autorise 100 requêtes par fenêtre de 60 secondes, il n'en reste aucune et le quota revient dans 20 secondes. Si les deux sont présents, le projet donne la priorité à `Retry-After`. Un client bien conçu attend au moins ce délai, ajoute un petit délai aléatoire et, en l'absence de `Retry-After`, utilise une temporisation exponentielle plutôt qu'un intervalle fixe.

Les 22 limiteurs de débit conçus pour refuser le trafic en cas de panne, réalisés pour un annuaire public d'entreprises et de professionnels dans le cadre d'un poste occupé depuis 2025, suivent cette règle par chemin : chacun a été construit pour le chemin qu'il protège, car une tentative de connexion, une recherche et un export massif deviennent abusifs à des fréquences très différentes. Leur état réside dans Redis, afin que la limite tienne sur toutes les instances plutôt que d'être propre à chaque processus. Le choix décisif porte sur la panne de Redis : les limiteurs refusent le trafic au lieu de le laisser passer. La page expose franchement le prix de ce choix, qui rend un problème Redis visible des utilisateurs, et sa justification : l'autre option serait qu'un problème Redis devienne un problème de facturation. Cloudflare Turnstile, un verrouillage de l'origine et 351 lignes de middleware d'idempotence, empêchant qu'une écriture réessayée soit appliquée deux fois, les entourent.

Où les problèmes surviennent

Chaque serveur tient son propre compteur. Une limite conservée dans la mémoire d'un processus se multiplie avec chaque copie de l'application : quatre instances derrière un répartiteur autorisent quatre fois le débit prévu, et chaque redémarrage remet les compteurs à zéro. La zone de nginx est une mémoire

partagée sur une machine, et la synchronisation des zones entre serveurs fait partie de son abonnement commercial ; le protocole peers de HAProxy copie les entrées des tables de suivi entre instances, mais, comme le précise le manuel, sans les additionner. Conservez les compteurs dans un stockage consulté par toutes les instances, comme l'annuaire ci-dessus avec Redis, ou appliquez la limite au point unique que traverse tout le trafic.

Tout le monde arrive de la même adresse. Derrière un proxy inverse, un CDN ou un répartiteur de charge, l'application voit une seule adresse cliente pour tous les utilisateurs si l'adresse d'origine n'est pas transmise, et une limite par IP ralentit alors tout le monde à la fois. Un bureau entier ou un réseau mobile derrière NAT produit le même effet de façon moins marquée. Utilisez comme clé l'adresse réelle du client, issue d'un en-tête que seul votre propre proxy peut définir, et préférez le compte ou la clé d'API à l'adresse pour le trafic authentifié.

Le limiteur signale une panne du serveur. La réponse par défaut de nginx à une requête rejetée est 503 Service Unavailable, qui signifie en HTTP que le serveur est surchargé ou indisponible pour maintenance. Les tableaux de bord et alertes qui traitent les réponses 5xx comme des erreurs serveur réveillent alors quelqu'un parce qu'un client s'est montré trop gourmand, sans que le client apprenne quoi que ce soit sur son propre comportement. Répondez avec 429, qui situe la cause côté client, et journalisez les rejets à un niveau distinct.

Tous les clients limités reviennent à la même seconde. Si un quota est réinitialisé à un instant fixe, ou si tous les clients refusés reçoivent le même `Retry-After`, ils reviennent ensemble ; le projet de l'IETF donne l'exemple d'une remise à zéro quotidienne à 18:00:00, lorsque tous les clients risquent d'arriver à 18:00:00. La rafale suivante peut être pire que celle qui a été limitée. Ajoutez de l'aléatoire des deux côtés : faites varier l'attente annoncée par le serveur et laissez les clients ajouter leur propre délai aléatoire.

Personne n'a décidé quoi faire en cas de panne du stockage. Un limiteur qui conserve ses comptes dans Redis ou une base doit réagir lorsque ce stockage est injoignable. Un code qui intercepte l'erreur de connexion et laisse passer la requête choisit de rester ouvert en cas de panne, ce qui transforme l'interruption en période sans limite pour qui la surveille ; un code qui refuse choisit de rester fermé, ce qui en fait une interruption pour tous. Aucune des deux options n'est mauvaise dans tous les cas, mais le choix relève d'une décision de conception par chemin, comme pour l'annuaire ci-dessus, et non du comportement par défaut d'une bibliothèque.

Compter les refus révèle des informations. Si les réponses 401 et 403 sont décomptées du quota, prévient le projet de l'IETF, un client malveillant pourrait sonder un point d'accès pour découvrir le trafic d'un autre utilisateur. Décidez quelles réponses comptent, donnez aux échecs d'authentification leur propre limite et ne communiquez pas les détails des quotas aux clients non authentifiés.

Pour aller plus loin

- RFC 6585 : codes d'état HTTP supplémentaires — définit 429 Too Many Requests et précise ce que le code laisse au serveur : l'identité de l'utilisateur et le comptage des requêtes.
- RFC 9110 : sémantique HTTP — le champ `Retry-After`, ses deux formes et le sens de 503 lorsque le serveur le renvoie.
- Champs d'en-tête RateLimit pour HTTP — le projet du groupe de travail HTTPAPI de l'IETF sur `RateLimit-Policy` et `RateLimit`, avec une section de sécurité sur les retours synchronisés et les fuites d'information.
- Module ngx_http_limit_req_module — `limit_req`, `burst`, `nodelay`, `delay`, le mode de simulation et les tailles de la zone de mémoire partagée.
- Manuel de configuration de HAProxy 3.4 — les tables de suivi et leurs compteurs de débit, et le protocole peers qui les copie entre instances.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que la limitation de débit ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/rate-limiting/